

A numerical analysis of Quicksort: How many cases are bad cases?

Guido Hartmann

*Kiel University of Applied Sciences, Germany**

July 15, 2015

ABSTRACT

We present numerical results for the probability of bad cases for Quicksort, i. e. cases of input data for which the sorting cost considerably exceeds that of the average. Dynamic programming was used to compute solutions of the recurrence for the frequency distributions of comparisons. From these solutions, probabilities of numbers of comparisons above certain thresholds relative to the average were extracted. Computations were done for array sizes up to $n = 500$ elements and for several methods to select the partitioning element, from a simple random selection to what we call “recursive median of three medians.” We found that the probability strongly depends on the selection method: for $n = 500$ and a threshold 25% above the average number of comparisons it ranges from $2.2 \cdot 10^{-3}$ to $3.0 \cdot 10^{-23}$. A version of Quicksort based on the recursive median of medians approach is proposed, for which our data suggest a worst case time complexity of $O(n^{1.37})$.

KEYWORDS: Quicksort; numerical analysis; bad cases; probability; frequency distribution of comparisons; recursive median of three medians; implementation

1. Introduction

In 1961, C. A. R. Hoare published a sorting method which he called “Quicksort” [1, 2]. The name is not an exaggeration: a careful implementation of Quicksort performs, in the average case and for large numbers of elements, better than any other of the established comparison based array sorting algorithms. A further advantage is that the sorting is done in place.

However, there remains the annoyance that the time complexity of Quicksort is actually $O(n^2)$, and $O(n \log n)$ only on average. For instance, sorting of one million elements may need a factor of 18,000 longer than expected if, unfortunately, a very “bad case” for the algorithm is encountered.

For the practitioner who has to make judgement on the usability of a Quicksort variant for a given application it would be desirable to know probability values, at least orders of magnitude, for such cases. What is, for a given number of elements, the relative number of cases needing, say, more than 150% of the average sorting time? How does such a ratio depend on the number of elements, how on the threshold factor relative to the average, how on the Quicksort version used?

As yet, theoretically derived formulas for the probability of bad cases ([3, 4, 5]) hardly meet the expectations of the practitioner. Either it is necessary to fit a parameter to simulation results first, or only a bound is given that may be orders of magnitude above realistic values.

In this article an attempt is made to give answers to those questions, based only on numerical computations. We present results for the probability of cases for which the cost will be above certain thresholds relative to the average. As criterion for the cost we confine ourselves to the number of comparisons of elements, the basic operation that determines the time complexity of Quicksort. In addition, we restrict ourselves to arrays of pairwise different elements. However, we attach importance to base our computations on algorithms which are practically usable, and in particular perform satisfactorily also if elements of equal values are present.

*Fachhochschule Kiel – University of Applied Sciences, Fachbereich Informatik und Elektrotechnik, Grenzstraße 5, 24149 Kiel, Germany. E-mail: guido.hartmann@fh-kiel.de

To get our results, we numerically computed the complete frequency distributions of comparisons for up to 500 elements. We also studied the influence of different methods for the selection of partitioning elements on the probability of bad cases. As a by-product of our investigations, we are able to propose a Quicksort version that has extremely small probability values for bad cases and a time complexity below $O(n^2)$.

Frequency distributions of comparisons have already been computed by Eddy and Schervish [6] for numbers of elements up to 132, and by McDiarmid and Hayward [4] for 100 elements; in both cases, however, without extracting the probabilities we are interested in.

The remainder of this paper is organized as follows. We give a short recollection of the Quicksort algorithm and show the partitioning method on which our computations are based in Section 2. Following in Section 3, we present the fundamental formulas, particularly for the determination of frequency distributions. After this, in Section 4 we discuss the models and approximations we used, including the recursive median of three medians approach. In Section 5, some technical details of the computations are given. The results are presented in Section 6. Following in Section 7, characteristics of several methods for partitioning are discussed. In Section 8 we comment on our Quicksort implementation. Finally, Section 9 concludes the paper. A listing of our proposed Quicksort version is given in the appendix.

Note that throughout this article variable name n will exclusively be used for the number of array elements to be sorted.

2. Quicksort fundamentals

2.1 The basic algorithm

Quicksort is a recursive algorithm following the divide-and-conquer paradigm. It may be formulated as follows:

Recursion basis:

If $n \leq 1$, nothing has to be done.

Recursive part (if $n > 1$):

Choose one of the array elements as “partitioning element” p . Partition the array, i. e. rearrange its elements so that an order

$$\underbrace{\text{elements} \leq p}_{\text{subarray 1}}, \text{ one element} = p, \underbrace{\text{elements} \geq p}_{\text{subarray 2}} \quad (1)$$

or

$$\underbrace{\text{elements} \leq p}_{\text{subarray 1}}, \underbrace{\text{elements} \geq p}_{\text{subarray 2}} \quad (2)$$

is achieved. Only in the case of arrangement (1) one of the subarrays may be empty. Sort each of the two subarrays by recursion.

There are variants in which small subarrays are sorted by a simpler method; in this case the recursion basis has to be extended. Two details have been left out in the formulation above:

1. The method of choosing the partitioning element.
2. The partitioning method.

Our computations have been carried out for several methods of choosing the partitioning element, these will be discussed in Section 4. Point 2 will be dealt with in the following subsection.

```

/* ipart: index of the partitioning element */
partval = a[ipart];
i = 0;
j = n-1;
a[ipart] = a[j]; a[j--] = partval;
done = FALSE;
do {
    while (a[i] < partval)
        i++;
    while (i < j && partval < a[j])
        j--;
    if (j <= i)
        done = TRUE;
    else {
        temp = a[i]; a[i++] = a[j]; a[j--] = temp;
    }
} while (!done);
a[n-1] = a[i]; a[i] = partval;

/* subarray 1: a[0..(i-1)], subarray 2: a[(i+1)..(n-1)] */

```

Figure 1. Partitioning algorithm. FALSE and TRUE are symbols for 0 and 1, respectively.

2.2 Algorithm for partitioning

For our computations we assumed a partitioning method similar to that given by Sedgewick [7], program 7.2. Figure 1 shows our algorithm, formulated in C. An explanation for the choice of this version will be given in Section 7. The result is always arrangement (1) as displayed in Section 2.1, where one of the subarrays may be empty. The number of comparisons needed is either $n-1$ or n . For distinct values of elements, it turns out that if the final index of the partitioning element is i , the respective probabilities $q_n^{(c)}(i)$ for the numbers of comparisons c are

$$q_n^{(n-1)}(i) = \frac{n-1-i}{n-1} \quad \text{and} \quad q_n^{(n)}(i) = \frac{i}{n-1}. \quad (3)$$

3. Fundamental formulas

Our computations are based on the assumption that partitioning within Quicksort is done by the algorithm shown in Figure 1. For pairwise different elements, this will always lead to an arrangement

$$\underbrace{\text{elements} < p, p}_{\text{subarray 1}}, \underbrace{\text{elements} > p}_{\text{subarray 2}}. \quad (4)$$

Throughout this and the following section, variable name i will be used exclusively for the index of the partitioning element p after the partitioning process (indices numbered from 0).

Let f_n be the distribution of the frequencies of comparisons needed by Quicksort to sort arrays of n pairwise different elements ($n \geq 0$). Precisely: counting over all permutations, $f_n(j)$ is the number of cases for which j comparisons of elements are needed. So f_n is a function

$$f_n : \mathbb{N}_0 \longrightarrow \mathbb{N}_0 \quad \text{satisfying} \quad \sum_{j \geq 0} f_n(j) = n! \quad (5)$$

(\mathbb{N}_0 : the set of natural numbers including 0). The distributions may be computed from the recurrence

$$f_0(0) = f_1(0) = 1, \quad (6)$$

$$f_0(j) = f_1(j) = 0 \quad \text{for } j > 0, \quad (7)$$

$$f_n = d_n * \frac{1}{2}(\delta_{n-1} + \delta_n) * \sum_{i=0}^{n-1} \binom{n-1}{i} p_n(i) f_i * f_{n-1-i} \quad \text{for } n > 1. \quad (8)$$

Equations (6) and (7) define the basis and express the fact that cases $n = 0$ and $n = 1$ do not need any comparison. In equation (8) the recurrence for the distribution f_n as a whole is formulated. Here p_n is a function such that $p_n(i)/n$ means the probability that the element chosen for partitioning will be at index position i after the rearrangement. We assume the symmetry $p_n(i) = p_n(n-1-i)$ and require

$$\sum_{i=0}^{n-1} p_n(i) = n, \quad (9)$$

so that for the case of randomly chosen partitioning elements $p_n(i)$ is 1 for $0 \leq i < n$ and thus could be left out. The binomial coefficient takes numbers of cases according to equation (5) into account.

Symbol d_n denotes the distribution of the numbers of comparisons that are needed only for the selection of a partitioning element. In detail: $d_n(j)$ is the probability of needing j comparisons of elements to find the partitioning element. This function is normalized as

$$\sum_{j \geq 0} d_n(j) = 1. \quad (10)$$

For any $k \in \mathbb{N}_0$, function δ_k is defined as

$$\delta_k : \mathbb{N}_0 \longrightarrow \mathbb{N}_0 \quad \delta_k(j) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{for } j = k, \\ 0 & \text{for } j \neq k. \end{cases} \quad (11)$$

The result of a convolution of δ_k with another function will be this function shifted by k . In our recurrence equation, expression $(\delta_{n-1} + \delta_n)/2$ addresses the comparisons needed for the partitioning process itself. Symmetry has been used to derive this simpler expression from the original $(q_n^{(n-1)}(i) \delta_{n-1} + q_n^{(n)}(i) \delta_n)$ within the sum (cf. equations (3)).

Finally, symbol $*$ in the recurrence equation represents the convolution operator, i. e. for functions $g, h : \mathbb{N}_0 \longrightarrow \mathbb{R}$

$$(g * h)(j) \stackrel{\text{def}}{=} \sum_{k=0}^j g(k) h(j-k) \quad \text{for } j \geq 0. \quad (12)$$

The average number (expected value) of comparisons needed by Quicksort

$$\bar{C}_n \stackrel{\text{def}}{=} \frac{1}{n!} \sum_{j \geq 0} j f_n(j) \quad (13)$$

may be computed separately from the recurrence

$$\bar{C}_n = \bar{C}_n^{\text{select}} + n - \frac{1}{2} + \frac{2}{n} \sum_{i=0}^{n-1} p_n(i) \bar{C}_i \quad \text{for } n > 1, \quad (14)$$

where $\bar{C}_n^{\text{select}}$ is the average number of comparisons to select a partitioning element, i. e.

$$\bar{C}_n^{\text{select}} \stackrel{\text{def}}{=} \sum_{j \geq 0} j d_n(j). \quad (15)$$

For the maximum number of comparisons

$$\widehat{C}_n \stackrel{\text{def}}{=} \max_{j \geq 0 \wedge f_n(j) > 0} j \quad (16)$$

we have the recurrence

$$\widehat{C}_n = \widehat{C}_n^{\text{select}} + n + \max_{0 \leq i < n \wedge p_n(i) > 0} (\widehat{C}_i + \widehat{C}_{n-1-i}) \quad \text{for } n > 1, \quad (17)$$

where $\widehat{C}_n^{\text{select}}$ denotes the maximum number of comparisons to select a partitioning element.

4. Models and approximations

In recurrence (8), functions p_n and d_n have yet to be fixed. We did our computations for several methods of selecting the partitioning element; the two functions depend on the respective method. Additionally, we analyzed an extended version of the Quicksort algorithm where an insertion sort is used for small arrays. These different versions and the respective approximations used in the computations will be classified as five models, described in the following.

We are interested in the frequencies of bad cases, which result from summations of large numbers of comparisons during traversal of the recursion tree for recurrence (8). Any approximations we used in the computations were chosen as such that rather some overestimation than underestimation of these cases could occur.

4.1 Model 1: Simple

In the simplest version, the partitioning element for an array of n elements is always taken from a fixed position, usually at index $\lfloor n/2 \rfloor$. Since we are counting over all permutations of n pairwise different elements, this is equivalent to a random choice where each element has the same probability of being selected. Hence we have

$$p_n(i) = 1 \quad \text{for } 0 \leq i < n. \quad (18)$$

No comparisons are needed for this simple selection, so the convolution with d_n can simply be left out.

4.2 Model 2: Median of three

Quicksort performs most efficiently if the two subarrays which result from partitioning are equal (or almost equal) in size. Therefore selection methods have been invented to increase the probability of the partitioning element to end up at a position near the middle of the array.

Hoare [2] suggested to take a small sample of array elements and use its median for partitioning. In 1969, Singleton [8] published a version that incorporates method “median of three elements” (or “median of three” for short). From the first, middle and last element in the array the median valued one is chosen. In contrast to Singleton’s version, we assume that the median is determined without an explicit sort among the three elements. According to our model, method “simple” is used for $n = 2$. For $n \geq 3$, median of three leads to

$$p_n(i) = a_n i(n-1-i) \quad \text{for } 0 \leq i < n, \quad (19)$$

where

$$a_n = n / \binom{n}{3} = \frac{6}{(n-1)(n-2)}. \quad (20)$$

Figure 2a shows this function for $n = 500$ together with the result of a simulation made with one million random permutations of n pairwise different elements.

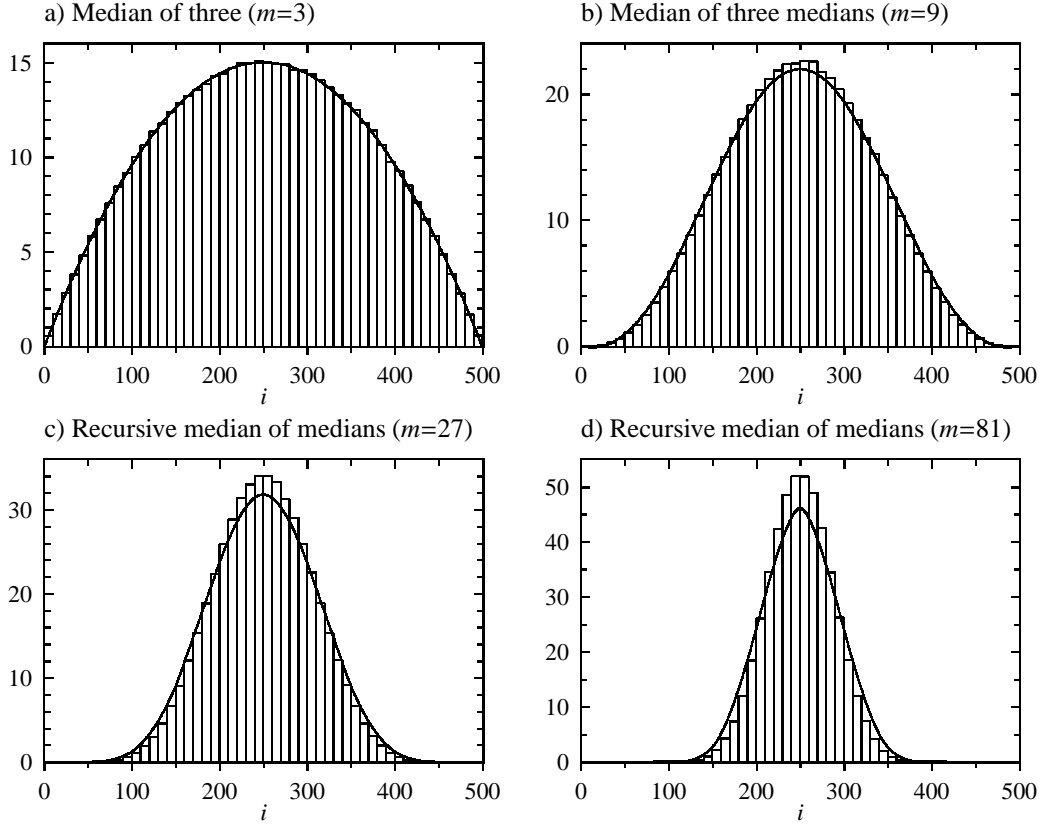


Figure 2. Simulation results and model functions for p_n ($n = 500$). Parameter m is the sample size. Frequencies of indices of partitioning elements are counted in bins of width 10. The ordinate scales correspond to frequencies, transformed according to equation (9), within bins. Correspondingly, the model functions have been multiplied by the bin width.

With this method, a selection of the partitioning element is not free of charge. For d_n , $n \geq 3$, we have

$$d_n(2) = \frac{1}{3}, \quad d_n(3) = \frac{2}{3}, \quad d_n(j) = 0 \text{ otherwise.} \quad (21)$$

To save the convolution with this function, we used the approximation

$$d_n(3) = 1, \quad d_n(j) = 0 \text{ otherwise} \quad (22)$$

instead, i. e. we assumed always three comparisons for the selection process.

4.3 Model 3: Median of three medians

In their article on a new implementation of the C library function `qsort`, Bentley and McIlroy [9] showed the usage of selection method “median of three medians, each of three elements” (or simply “median of three medians”) for array sizes n above some threshold value. A set of nine sample elements is divided into three equally-sized subsets. The median within each subset is determined, and the median of these three medians is used for partitioning.

The probability function for the final position of partitioning elements, resulting from this method, has been given by Durand [10]. Rewritten according to our conventions, it reads ($n \geq 9$, $0 \leq i < n$):

$$p_n(i) = b_n \left(3 \prod_{j=0}^2 (i-j) \prod_{j=0}^4 (n-1-i-j) + 10 \prod_{j=0}^3 (i-j)(n-1-i-j) + 3 \prod_{j=0}^4 (i-j) \prod_{j=0}^2 (n-1-i-j) \right), \quad (23)$$

where

$$b_n = 36 / \prod_{j=0}^7 (n-1-j). \quad (24)$$

In our computations, we used an approximation of this function instead, mainly for consistency with those in models 4 and 5, namely

$$p_n(i) = a_n i(i-1)(i-2)(n-1-i)(n-2-i)(n-3-i), \quad (25)$$

where factor a_n was computed numerically. This is the simplest polynomial that has the correct zeros at index positions 0, 1, 2, $n-1$, $n-2$, $n-3$. The polynomial given in equation (23) has two additional zeros at argument values

$$\frac{n-1}{2} \pm \frac{1}{2} \sqrt{4n^2 - 59n + 217}. \quad (26)$$

Our approximation slightly underestimates the concentration effect to the middle of the array as can be seen in Figure 2b, where it is shown together with the result of a simulation, again made with one million random permutations.

The determination of the four medians of three may be interpreted as a sequence of Bernoulli trials, each of them with probability $1/3$ for the “success” of needing only two comparisons. The corresponding binomial distribution, transformed to the distribution of the total number of comparisons for the selection, leads to

$$d_n(j) = \binom{4}{j-8} \left(\frac{1}{3}\right)^{12-j} \left(\frac{2}{3}\right)^{j-8} \quad \text{for } 8 \leq j \leq 12, \quad 0 \text{ otherwise.} \quad (27)$$

Again, we avoided the convolution with this function for efficiency reasons and assumed the maximum value of 12 comparisons for each selection, in accordance with the principle given at the beginning of this section.

4.4 Model 4: Recursive median of three medians

The selection method of this model (“recursive median of medians” for short) is an extension of the methods from models 2 and 3 which suggests itself. It contains these two as special cases. For a given sample of m elements, m a power of 3 and ≥ 3 , the method may be formulated as a recursive algorithm:

Recursion basis:

If $m = 3$, determine the median of the three elements.

Recursive part (if $m \geq 9$):

Divide the set of sample elements into three subsets, each of size $m/3$. Determine the recursive median of medians element for each of these subsets by recursion. Then determine the median of these three elements.

If the sample is drawn from a set of pairwise different elements, there must always be

$$i_{\min} = 2^k - 1, \quad \text{where } k = \log_3 m, \quad (28)$$

elements less than and i_{\min} elements greater than the partitioning element. So we can infer

$$p_n(i) = 0 \quad \text{for } 0 \leq i < i_{\min} \text{ and for } n - i_{\min} \leq i < n. \quad (29)$$

As an approximation for p_n we used the simplest polynomial that has zeros at the correct index positions i , this is

$$p_n(i) = a_n \prod_{j=0}^{i_{\min}-1} (i-j)(n-1-i-j). \quad (30)$$

For $m \geq 9$, factor a_n was computed numerically.

Figures 2c and 2d display this approximation for $m = 27$ and $m = 81$, respectively, again together with results of simulations with one million random permutations each. The sequence of Figures 2a to 2d illustrates the increasing concentration effect towards the middle of the array if m is increased. It also shows that our approximations increasingly underestimate this effect with increasing m .

The number of median of three determinations needed for a selection is

$$t = \frac{m-1}{2}. \quad (31)$$

So the examination that led to equation (27) can be generalized to sequences of t Bernoulli trials, which results in

$$d_n(j) = \binom{t}{j-2t} \left(\frac{1}{3}\right)^{3t-j} \left(\frac{2}{3}\right)^{j-2t} \quad \text{for } 2t \leq j \leq 3t, \quad 0 \text{ otherwise.} \quad (32)$$

Again, we used the maximum value of $3t$ comparisons instead of computing the convolution with this function.

To support the formation of well balanced recursion trees, we wish that sample sizes increase with n . So we choose m as the greatest power of 3 satisfying

$$q_{\min} \leq \frac{n}{m}, \quad (33)$$

where q_{\min} is an integer parameter ≥ 1 . This leads to a number of comparisons needed for the selection process that is bounded as

$$\widehat{C}_n^{\text{select}} \leq \frac{3}{2} \left(\frac{n}{q_{\min}} - 1 \right) \quad \text{for } n \geq 3q_{\min}, \quad (34)$$

i. e. by a linear function in n . Thus with this method Quicksort maintains its $O(n \log n)$ complexity for the average case.

An increase of the sample size also helps to prevent that “non-random” data, i. e. sequences already possessing some order, are preferred candidates for bad cases.

In the following, the term “recursive median of medians” will be used for this adaptive version where the sample size depends on n , unless specifically stated otherwise.

It should be mentioned that there exists a method to determine the k th smallest element—and thus also the median—of n elements that has worst case time complexity $O(n)$ (Blum et al. [11]). Its high cost, however, does not recommend its use within Quicksort.

4.5 Model 5: Recursive median of medians plus insertion sort

Already in 1962, Hillmore [12] remarked that the performance of Quicksort could be improved if $n = 2$ was treated as a basis case of the recursive algorithm. Hibbard [13] suggested to extend the basis further and use Shellsort for small arrays. In the program version of Singleton [8], Shellsort was replaced by insertion sort, which has become standard since then. For this method (“sorting by straight insertion”), the frequency distributions of comparison may be computed from the recurrence

$$f_0(0) = f_1(0) = 1, \quad (35)$$

$$f_0(j) = f_1(j) = 0 \quad \text{for } j > 0, \quad (36)$$

$$f_n = \sum_{k=1}^{n-1} (1 + \delta_{n-1}(k)) \delta_k * f_{n-1} \quad \text{for } n > 1. \quad (37)$$

The average number of comparisons needed by insertion sort is

$$\overline{C}_n = \frac{n(n+3)}{4} - H_n, \quad (38)$$

where H_n is the n th harmonic number $\sum_{k=1}^n k^{-1}$.

For the maximum number of comparisons we have

$$\widehat{C}_n = \frac{n(n-1)}{2}. \quad (39)$$

In model 5, we extended model 4 so that equation (37) holds for values of n up to a parameter value n_{\max}^b and equation (8) for $n > n_{\max}^b$. It turned out that up to $n = 9$, the maximum number of comparisons needed for insertion sort is not greater than the respective value for models 2 to 4. So for our computations we used $n_{\max}^b = 9$ for model 5.

5. Technical details of computations

Most of our numerical computations were done on a Sun Fire X4270 M2 server machine with x86 architecture, running under Solaris 10. The recurrence equations for the average numbers (14) and maximum numbers (17) of comparisons were transformed into recursive programs—written in C—with dynamic programming. Thus most invocations of the respective recursive functions to get a result simply lead to a table lookup.

Our main development task was an implementation to solve recurrence (8) (plus recurrence (37) in the case of model 5) for the frequency distributions of comparisons. During program execution, for all distribution functions f_k computed so far the frequency values within the functions’ supports are stored in a large array accompanied by an integer array containing limits of the supports and pointers. Thus every distribution had to be computed only once. Wherever possible, use has been made of symmetries in recurrence equations (8), (14), and (17).

As element type for the tables needed to implement dynamic programming we used the extended precision floating point type (long double in C). On the x86 machine, it has a relative machine accuracy of 10^{-19} and a maximum value of about 10^{4932} . We took care to do the summations in a way that rounding errors are kept small. Crosscheck runs on a Sun Fire 280R machine with UltraSPARC III processors and a relative machine accuracy of $2 \cdot 10^{-34}$ for type long double did not show any difference in the results as printed with at least four significant decimal digits.

Unfortunately, application of the convolution theorem with fast Fourier transform works only for rather small values of n . For example, with model 1 and $n \gtrsim 40$ on the x86 machine and $n \gtrsim 53$ on the SPARC machine, rounding errors in Fourier coefficients lead to results for the probability of bad cases which differ from those computed using equation (12) directly. With increasing n , the tails of distributions get more and more transformed into pure rounding noise if the convolution theorem is applied.

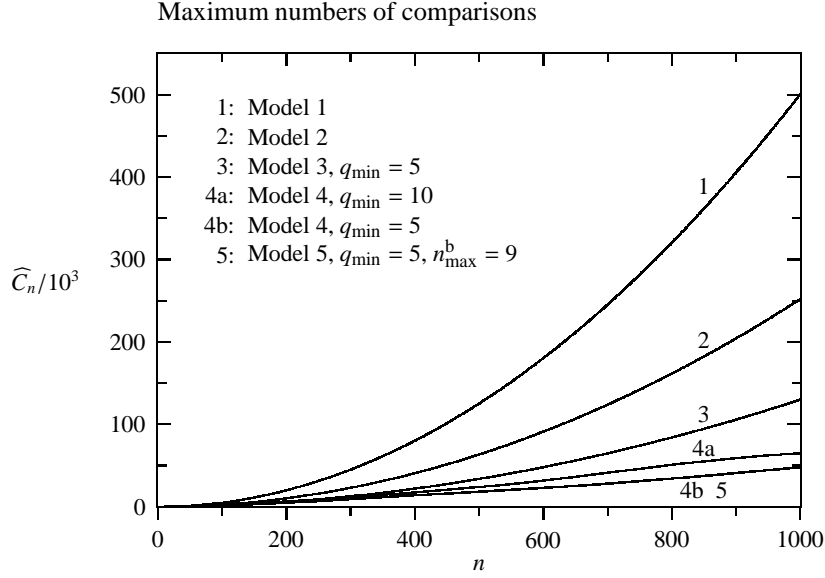


Figure 3. Maximum numbers of comparisons. Curves 4b and 5 are visually indistinguishable.

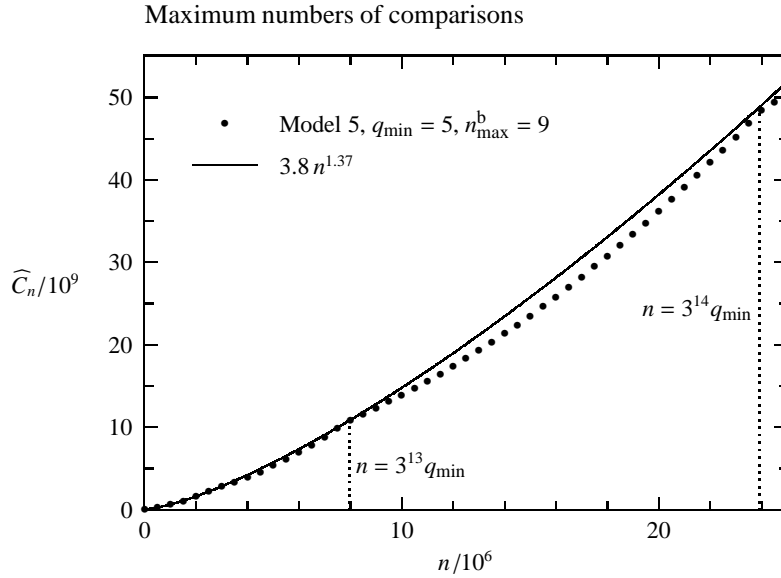


Figure 4. Maximum numbers of comparisons for model 5, displayed for values of n up to 25 millions. Two thresholds for sample sizes are indicated by dotted lines. Additionally, the figure shows the graph of a function that may be an asymptotical upper bound for \widehat{C}_n .

6. Results

We first present numerical results for worst and average cases, i. e. results that could be obtained without the need to compute frequency distributions of comparisons. Afterwards, we show results for standard deviations of comparisons and probabilities of bad cases.

Parameter name q_{\min} , defined for models 4 and 5 (equation (33)), will also be used for model 3, meaning here that selection method median of three medians is used for arrays of size $n \geq 9q_{\min}$, and method median of three elements below this threshold.

6.1 Worst case

Figure 3 shows the maximum numbers of comparisons for several models and parameters up to $n = 1000$, computed from the recurrence equation (17).

If the recursive median of medians method with a fixed sample size is used for selection of the partitioning element, the worst case complexity of Quicksort is $O(n^2)$. In particular: for a sample size $m = 3^k$, $k \geq 0$, it is theoretically expected that the maximum number of comparisons is

$$\widehat{C}_n = 2^{-(k+1)} n^2 + \text{lower order.} \quad (40)$$

Curves 1, 2, and 3 in Figure 3 can be regarded as compatible with the expectation of leading terms $n^2/2$, $n^2/4$, $n^2/8$, respectively. For models 4 and 5, however, the increase of the curves appears much slower than n^2 . This is not surprising, because the sample size is stepwise increasing with n . The fact that curves 4b and 5 (models 4 and 5 with the same value for q_{\min}) are visually indistinguishable shows that the influence of using insertion sort for small arrays is negligible for the worst case, as could be expected from our choice of n_{\max}^b (Section 4.5).

We also used McIlroy's "killer adversary" [14] to construct bad cases for our Quicksort versions. The adversary program was applied to models 1 to 5, with $q_{\min} = 5$ for models 3 to 5 and $n_{\max}^b = 9$ for model 5. The following numbers of comparisons were determined:

n	Model 1	Model 2	Model 3	Model 4	Model 5
100,000	$2.500 \cdot 10^9$	$2.500 \cdot 10^9$	$8.338 \cdot 10^8$	$2.848 \cdot 10^6$	$2.768 \cdot 10^6$
200,000	$1.000 \cdot 10^{10}$	$1.000 \cdot 10^{10}$	$3.334 \cdot 10^9$	$5.922 \cdot 10^6$	$5.760 \cdot 10^6$
500,000	$6.250 \cdot 10^{10}$	$6.250 \cdot 10^{10}$	$2.084 \cdot 10^{10}$	$1.515 \cdot 10^7$	$1.474 \cdot 10^7$
1,000,000	$2.500 \cdot 10^{11}$	$2.500 \cdot 10^{11}$	$8.334 \cdot 10^{10}$	$3.100 \cdot 10^7$	$3.016 \cdot 10^7$

These data, too, suggest that for models 4 and 5 the increase is much slower than the $O(n^2)$ behavior of models 1 to 3, even though the adversary program assumes $O(1)$ complexity for the selection of the partitioning element whereas it is actually $O(n)$ for those two models (equation (34)).

To get an estimate for the complexity of our recursive median of medians models (4 and 5), we computed the maximum numbers of comparisons for model 5 up to $n = 25$ millions. The result is shown in Figure 4. In addition to our data, we display a function

$$b(n) = 3.8 n^{1.37} \quad (41)$$

which was determined such that it might be an asymptotical upper bound for \widehat{C}_n .

6.2 Average case

Average numbers of comparisons were computed from the recurrence equation (14). To keep Figure 5 readable, results are shown only for four of the six model/parameter combinations that were displayed in Figure 3. Additionally, we list average numbers for all models and for several values of n up to 10,000 (model identifiers corresponding to the numbering of curves in Figure 3):

n	Model 1	Model 2	Model 3	Model 4a	Model 4b	Model 5
1000	11,319	10,884	10,704	10,713	10,997	10,394
2000	25,396	24,134	23,590	23,564	24,376	23,171
5000	72,630	68,171	66,192	66,232	69,039	66,027
10000	159,105	148,211	143,305	143,578	149,187	143,165

Models 2 and 3 show smaller values of averages compared to model 1. The dependence of the average on n is known to be

$$\overline{C}_n = a n \log_e n + \text{lower order,} \quad (42)$$

where a is 2 for model 1 (Knuth [15]), $12/7$ for model 2 (Sedgewick and Flajolet [16]), and $12600/8027$ for model 3 (Durand [10]). So the reductions relative to model 1 are expected to be asymptotically

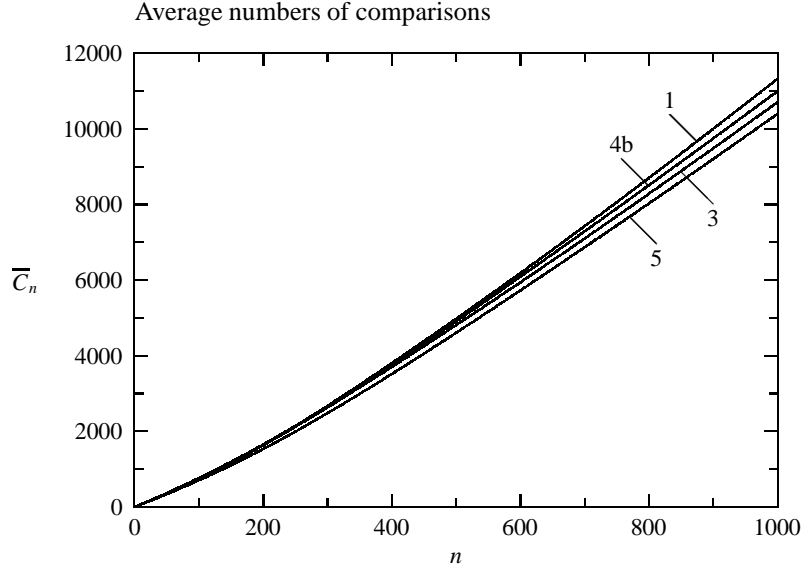


Figure 5. Average numbers of comparisons. Numbering of curves is the same as in Figure 3.

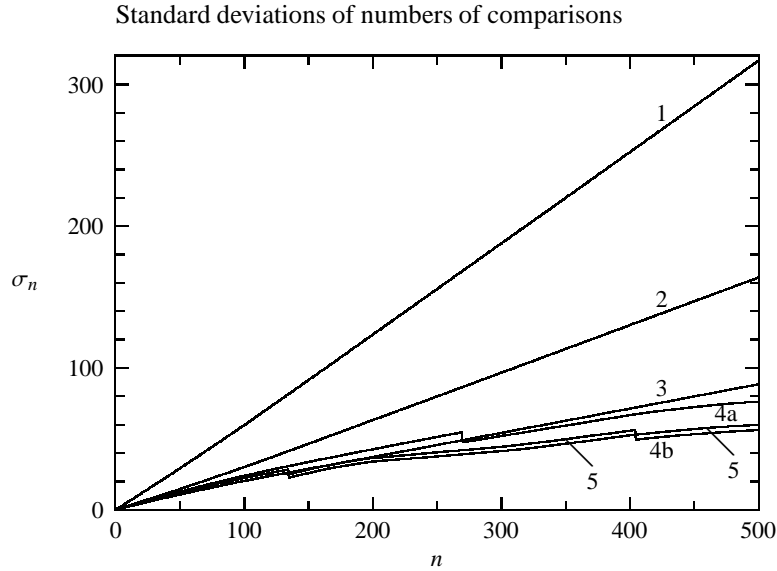


Figure 6. Standard deviations of numbers of comparison. Numbering of curves is the same as in Figure 3.

14.3% for model 2 and 21.5% for model 3. Our values of 6.8% and 9.9%, respectively, at $n = 10,000$ indicate that we are yet far below the asymptotic region where the $n \log n$ term dominates.

In the range up to $n = 10,000$, model 4, where the sample size for selection of the partitioning element increases stepwise with n , does not show a further reduction of average values. We have to consider that the ratio of comparisons needed for the selection of the partitioning element, relative to the cost of partitioning, is between (asymptotically) $1/(2q_{\min})$ and $3/(2q_{\min})$. So for $q_{\min} = 5$, the total cost (selection plus partitioning) will be between 10% and 30% above that for the partitioning alone. As our data indicate, this additional cost is largely compensated by the improved balancing of the recursion tree.

In the range between $n = 1000$ and $10,000$, the data for model 5 show a reduction relative to model 4b—due to the usage of insertion sort for small arrays—that is decreasing from 5.5% to 4.0%.

The decrease results from the fact that the share of work done in the leaf nodes of the recursion tree is decreasing with increasing n .

6.3 Standard deviation

To determine standard deviations for the numbers of comparisons, defined as

$$\sigma_n \stackrel{\text{def}}{=} \sqrt{\frac{1}{n!} \sum_{j \geq 0} (j - \bar{C}_n)^2 f_n(j)}, \quad (43)$$

we first had to compute frequency distributions f_n from the recurrence equation (8) (plus recurrence (37) in the case of model 5), which was done for values of n up to 500. Results are shown in Figure 6. For models 1 to 3, the dependence on n appears almost linear. For a simple version of Quicksort corresponding to our model 1, Iliopoulos and Penman [17] proved the formula

$$\sigma_n = \sqrt{7n^2 - 4(n+1)^2 H_n^{(2)} - 2(n+1)H_n + 13n}, \quad (44)$$

which is compatible with our data (at $n = 500$, the difference is less than 0.1%). For models 2 and 3, we find values that are lower by factors of approximately 1/2 and 1/4, respectively. So not only the maximum numbers of comparisons (Figure 3), but also the standard deviations appear to decrease by a factor 1/2 if the sample size is tripled.

For models 4 and 5, the increase with n is less than linear as could be expected because the sample size is stepwise increasing with n . The steps at $3^k q_{\min}$ are visible in the figure, too. Due to the fact that the frequency distribution of comparisons is broader for insertion sort than that of the—in the range $n \leq n_{\max}^b$ —competing Quicksort with median of three selection used in model 4b, model 5 shows greater values of the standard deviation. For a suitably chosen value of n_{\max}^b , however, the average number of comparisons for selection sort is smaller in this range of n , so model 5 remains an improvement in efficiency.

6.4 Probability of bad cases

Probabilities of bad cases were determined from frequency distributions of comparisons. In Figure 7, we show some of these distributions at $n = 500$. The figure illustrates both the stronger than exponential decrease in the tails of the distributions and the increase of concentration as result of a greater effort to select the partitioning element. Incidentally, for model 1 there is exactly one worst case, which arises if always the element of largest value is picked for partitioning. This leads to $(n+2)(n-1)/2$ comparisons.

What is a bad case of input data? An attempt to define such a case may be, for example, that it should be one that needs more than 1.5 times the average number of comparisons. This would, however, appear rather artificial, because for some applications even smaller deviations might be unacceptable whereas for others a greater factor might be tolerable. For this reason we introduce a parameter, the “threshold factor” τ , to discriminate between bad cases and acceptable ones.

As “threshold factor” or “relative threshold value” we define

$$\tau \stackrel{\text{def}}{=} C^{\text{thr}} / \bar{C}_n, \quad (45)$$

where C^{thr} is the absolute threshold value and \bar{C}_n the average number of comparisons. In Figure 8, we display results for the probability of cases needing more comparisons than those given by a value of τ , in dependence on n . So the probability displayed is

$$p_{n,\tau} \stackrel{\text{def}}{=} \frac{1}{n!} \sum_{j > \tau \bar{C}_n} f_n(j), \quad (46)$$

where f_n is the frequency distribution defined in Section 3.

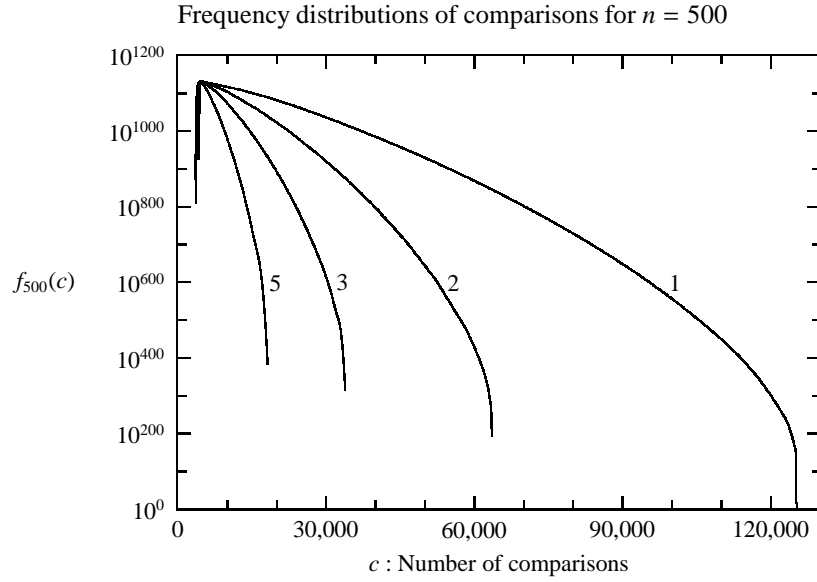


Figure 7. Frequency distributions of comparisons for $n = 500$. Curve labels are model numbers ($q_{\min} = 5$ for models 3 and 5, $n_{\max}^b = 9$ for model 5).

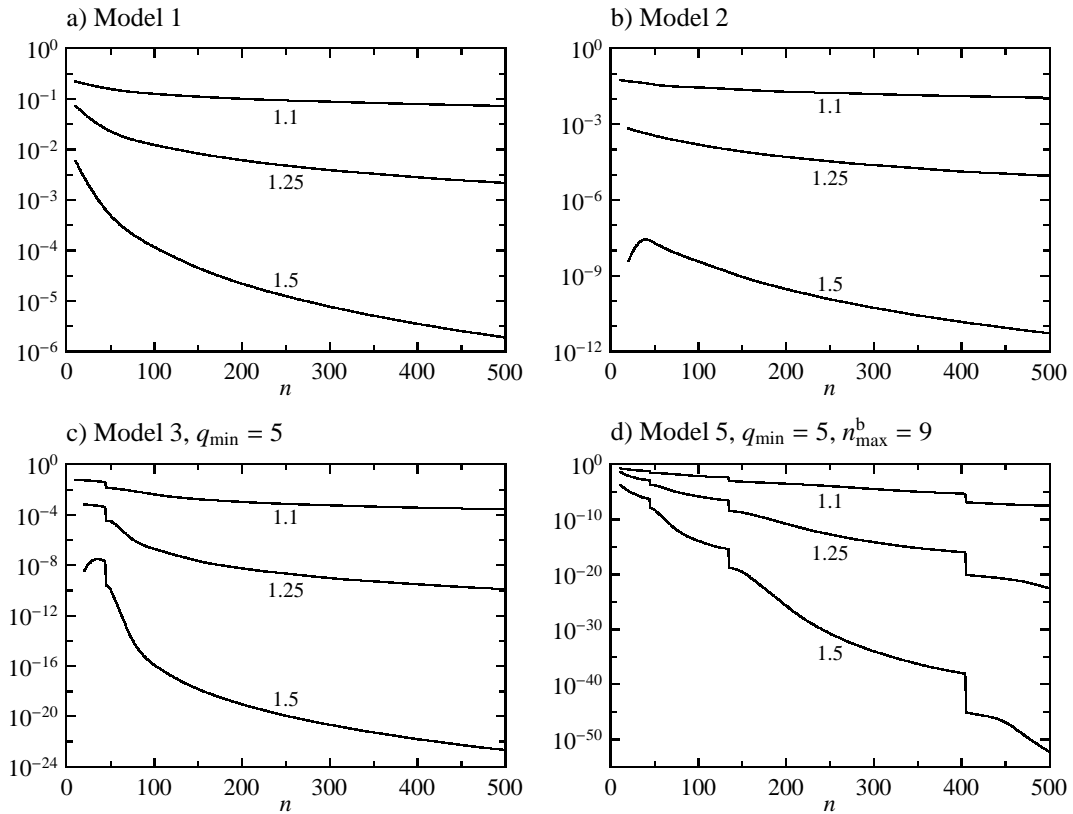


Figure 8. Relative numbers of cases needing more comparisons than the number given by a threshold factor τ , in dependence on n . Curve labels indicate the threshold factor.

For models 1, 2, 3, and 5, the figure shows function graphs of $p_{n,\tau}$ in dependence on n , for thresholds that are 10%, 25%, and 50% above the average number of comparisons. As a counterpart to this, Figure 9 displays probabilities for some values of n in dependence on τ . These two figures contain—for some given models, values of n and values of τ —an answer to the question posed in the title of this article.

The sequence of Figures 8a to 8d reveals the following facts:

1. For greater values of the threshold factor, the decrease of $p_{n,\tau}$ in dependence on n is stronger than that for smaller ones.
2. A greater sample size for the selection of the partitioning element leads to smaller probability values for bad cases and to a stronger decrease in n .
3. The adaptive method of model 5 leads to step effects at numbers of elements $n = 3^k q_{\min}$.

As a supplement to Figure 8 we present the following table which shows values for the ratio $p_{500,\tau}/p_{250,\tau}$:

τ	Model 1	Model 2	Model 3	Model 5
1.1	0.78	0.64	0.38	$4.44 \cdot 10^{-4}$
1.25	0.45	0.26	0.058	$1.60 \cdot 10^{-10}$
1.5	0.15	0.045	$1.89 \cdot 10^{-3}$	$2.16 \cdot 10^{-22}$
2.0	0.012	$9.24 \cdot 10^{-4}$	$2.63 \cdot 10^{-6}$	$1.90 \cdot 10^{-51}$

The table, too, indicates that the decrease of the relative number of bad cases with increasing n is strongly influenced by the value of τ and by the selection method.

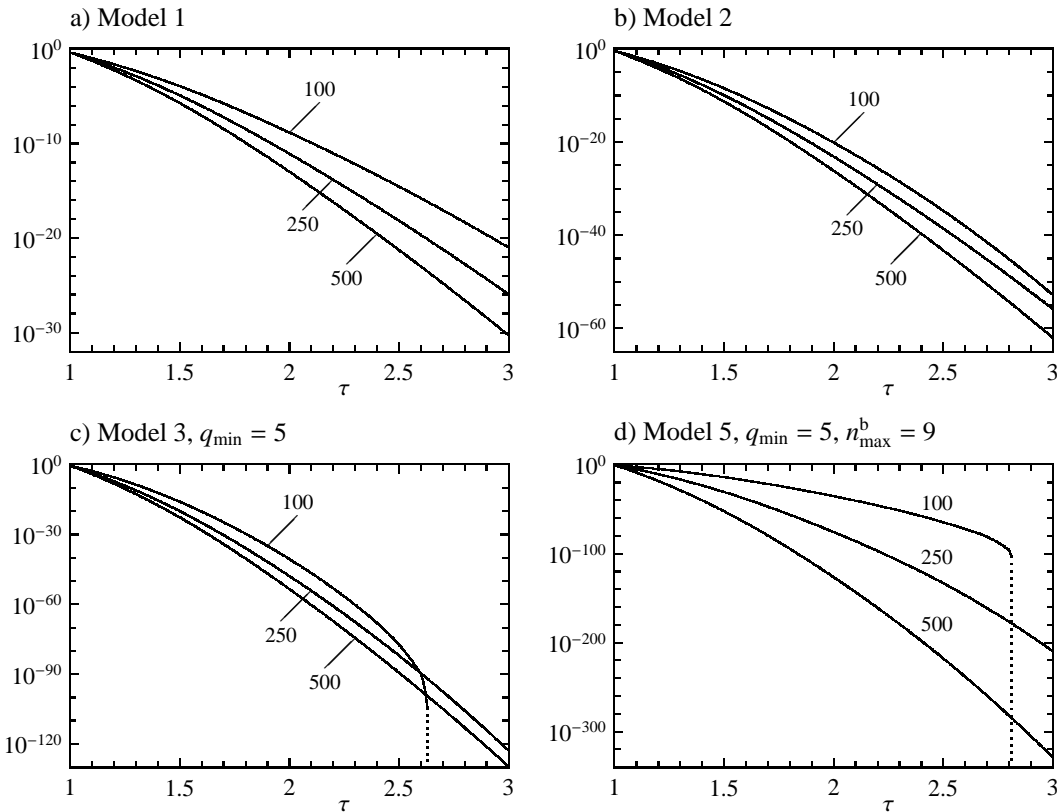


Figure 9. Relative numbers of cases needing more comparisons than the number given by a threshold factor for $n = 100, 250$, and 500 , in dependence on the threshold factor.

Figure 9 shows the probability of bad cases for $n = 100, 250$, and 500 in dependence on the threshold factor for values of τ up to 3 . The decrease of the probabilities with increasing τ turns out to be stronger than exponential as could be expected from the frequency distributions of comparisons (Figure 7). Even more than Figure 8, Figure 9 emphasizes the enormous differences in the order of magnitude among the different versions.

Additionally to these two figures, we list some probability values for bad cases at $n = 500$ in the following table:

τ	Model 1	Model 2	Model 3	Model 5
1.1	$7.35 \cdot 10^{-2}$	$1.14 \cdot 10^{-2}$	$2.83 \cdot 10^{-4}$	$3.75 \cdot 10^{-8}$
1.25	$2.17 \cdot 10^{-3}$	$8.88 \cdot 10^{-6}$	$1.28 \cdot 10^{-10}$	$2.97 \cdot 10^{-23}$
1.5	$1.88 \cdot 10^{-6}$	$5.10 \cdot 10^{-12}$	$2.17 \cdot 10^{-23}$	$5.12 \cdot 10^{-53}$
2.0	$1.02 \cdot 10^{-13}$	$6.66 \cdot 10^{-27}$	$3.62 \cdot 10^{-54}$	$4.25 \cdot 10^{-127}$

We try to support the judgement on these probability values in the following way. Assume that at intervals of one millisecond sorts of random data are started. Then we ask for the expected time until one event exceeding a given relative threshold value will occur. To give an example: for model 2 and $n = 500$, the expected time to encounter one case needing more comparisons than 1.25 times the average is 1.9 minutes. The following table presents such expected times for $n = 500$ (s: seconds, m: minutes, h: hours, d: days, a: years):

τ	Model 1	Model 2	Model 3	Model 5
1.1	0.014 s	0.09 s	3.5 s	7.4 h
1.25	0.46 s	1.9 m	90.5 d	$1.1 \cdot 10^{12}$ a
1.5	8.9 m	6.2 a	$1.5 \cdot 10^{12}$ a	$6.2 \cdot 10^{41}$ a
2.0	312 a	$4.8 \cdot 10^{15}$ a	$8.8 \cdot 10^{42}$ a	$7.5 \cdot 10^{115}$ a

Here it becomes clear that values for model 2, $\tau \geq 2$, model 3, $\tau \geq 1.5$, and model 5, $\tau \geq 1.25$ are beyond every imagination. For comparison: the age of our universe is “only” $1.4 \cdot 10^{10}$ years! And these are results for the rather small number $n = 500$. Probabilities will become even smaller if n is increased to values which are worth using the Quicksort algorithm for sorting.

It should be kept in mind, however, that these are results for uniformly distributed random data of distinct values. In fact, data that are not pairwise distinct are not a real problem if an adequate method for partitioning is used, as will be shown in Section 7. The main pitfall is the fact that data showing up in practice tend to possess some order pattern that might lead to a bad case. An attempt to solve this problem has been randomization as shown already in Hoare’s first articles on Quicksort [1]. On the other hand, this does not change the statistical probability for bad cases. Our adaptive recursive median of medians approach not only reduces the probabilities for such cases statistically to values that are virtually zero but also works against non-random data being preferred candidates for them.

7. Characteristics of partitioning algorithms

Partitioning algorithms may be divided into two categories, which we call “sweep methods” and “collision methods”, respectively, to have short expressive names at hand. With sweep methods, a scan is done by “sweeping” a pointer or index over the array from left to right. Collision methods use two pointers or indices, one for scanning from left to right, one from right to left, until they collide.

Simple sweep methods are shown in some textbooks, e. g. [18, 19]. Bentley [20] tells us that he learned such a scheme from Nico Lomuto. The algorithm displayed in Figure 10 produces the arrangement

$$\underbrace{\text{elements} < p}_{\text{subarray 1}}, \underbrace{p, \text{elements} \geq p}_{\text{subarray 2}}.$$

So elements equal to the partitioning element p (except p itself) will be placed into subarray 2. If all values of elements are equal, Quicksort needs the maximally possible number of $n(n-1)/2$ comparisons


```

/* ipart: index of the partitioning element */
partval = a[ipart];
a[ipart] = a[0];

i = 0;
for (j = 1; j < n; j++)
    if (a[j] < partval) {
        temp = a[++i]; a[i] = a[j]; a[j] = temp;
    }
a[0] = a[i];
a[i] = partval;

/* subarray 1: a[0..(i-1)], subarray 2: a[(i+1)..(n-1)] */

```

Figure 10. Partitioning algorithm (simple sweep method)

```

partval = a[ipart];

j = 0;
for (i = 0; i < n; i++)
    if (a[i] < partval) {
        temp = a[j]; a[j++] = a[i]; a[i] = temp;
    }

i = j;
for (k = i; k < n; k++)
    if (a[k] == partval) {
        temp = a[i]; a[i++] = a[k]; a[k] = temp;
    }

/* subarray 1: a[0..(j-1)], subarray 2: a[i..(n-1)] */

```

Figure 11. Partitioning algorithm (extended sweep method – three-way)

with this method. Since our objective is a general purpose version for sorting, a poor performance for “familiar” non-random cases is unacceptable: sequences which are, for example, (nearly) sorted (ascending or descending), have an organ pipe characteristic (as mentioned by Bentley and McIlroy [9]), or all equal values, should be processed at speeds not slower than the average. Hence we cannot consider this method as suitable for use in practice.

In an attempt to get a practically usable version, the algorithm may be extended to perform a “three-way partitioning” so that the arrangement

$$\underbrace{\text{elements} < p}_{\text{subarray 1}}, \text{elements} = p, \underbrace{\text{elements} > p}_{\text{subarray 2}} \quad (47)$$

is achieved. This can be done by two loops in sequence as shown in Figure 11.

Sweep methods are optimal with respect to comparisons. The algorithm of Figure 10 always needs $n-1$ comparisons, that of Figure 11 needs $(3n+1)/2$ comparisons on average (see also Tables 1 and 2 on page 20 for counting results). However, they have the disadvantage that elements may be moved several times until they reach their final position. Collision methods avoid this waste of resources: at the cost of a few additional comparisons they reduce the number of data movements—defined as

assignments in which one or both operands are array elements—asymptotically by a factor of three.

Already in his first series of papers on Quicksort, Hoare [1] published a collision algorithm for partitioning. In the following years, attempts were made to improve this method (Hibbard [13], Scowen [21], Singleton [8], van Emden [22]), of which Singleton’s version has become standard and was reproduced in various textbooks (e. g. Wirth [23, 24], Sedgewick [25], Cormen et al. [19]). Sometimes it is inaccurately attributed to Hoare. A listing of this algorithm, which we call the “classic collision method”, is shown in Figure 12.

It turns out that for arrays of n pairwise different elements, arrangement (4), shown in Section 3, is achieved with this method if and only if the partitioning element happens to be in its final position already from the beginning. In this case $n+1$ comparisons are needed. In all other cases arrangement (2), shown in Section 2.1, will result, and the number of comparisons is either n or $n+2$. For example, counting over all permutations of $n = 10$ elements (partitioning elements taken from index $\lfloor n/2 \rfloor$) yields the distribution

Comparisons:	10	11	12
Frequency:	756,000	362,880	2,509,920

where the $n+1 = 11$ comparisons occur in the $(n-1)!$ cases in which the partitioning element is in its final position already before the partitioning process. In order to always achieve arrangement (1) (Section 2.1), Scowen [21] suggested to swap the partitioning element, which he took from the middle of the array, to one end and swap it to its correct position afterwards. Sedgewick [26, 7] essentially combined this with Singleton’s version. However, in his simplest version of Quicksort he used the element already at the end of the array for partitioning, which leads to a worst case behavior for sorted sequences (ascending or descending) and gave rise to the rumor that this is a general property of Quicksort.

Our version (Figure 1), which we now call “new collision method”, has been derived from Sedgewick’s algorithm. For randomly chosen partitioning elements, all final index positions have equal probability, so equations (3) lead to the average number of $n-1/2$ comparisons. Sedgewick’s version needs $n+1-1/n$ comparisons on average.

Since the algorithm always produces arrangement (1), it is preferable to the classic collision method. Table 1 shows counting results for this version, too.

These algorithms can also be transformed into three-way versions by applying the two-pass strategy that was already shown for the sweep method. Figure 13 displays such a version for the classic collision algorithm. In their implementation of the C library function `qsort`, Bentley and McIlroy [9] used a more complicated technique, which needs more resources as can be seen in Table 2. In contrast to the extended classic collision algorithm, it may move elements back and forth even if the three-way arrangement (47) already exists. For `qsort`, however, it is the method of choice because it benefits from the three-way comparison results (less, greater, equal) which the user supplied comparison function has to deliver. Incidentally, it should be noted that the popular method to produce a three-way comparison result for integer operands by taking the difference may lead to arithmetic overflows for certain pairs of operand values; so it is not applicable within a library program.

```

partval = a[ipart];
i = 0;
j = n-1;
do {
    while (a[i] < partval)
        i++;
    while (partval < a[j])
        j--;
    if (i <= j) {
        temp = a[i]; a[i++] = a[j]; a[j--] = temp;
    }
} while (i <= j);

/* subarray 1: a[0..j],  subarray 2: a[i..(n-1)] */

```

Figure 12. Partitioning algorithm (classic collision method, Singleton [8])

```

partval = a[ipart];
j = 0;
k = n-1;
done = FALSE;
do {
    while (a[j] < partval)
        j++;
    while (j < k && partval <= a[k])
        k--;
    if (k <= j)
        done = TRUE;
    else {
        temp = a[j]; a[j++] = a[k]; a[k--] = temp;
    }
} while (!done);

k = j;
i = n-1;
done = FALSE;
do {
    while (k <= i && partval == a[k])
        k++;
    while (k <= i && partval < a[i])
        i--;
    if (i <= k)
        done = TRUE;
    else {
        temp = a[k]; a[k++] = a[i]; a[i--] = temp;
    }
} while (!done);

/* subarray 1: a[0..(j-1)],  subarray 2: a[(i+1)..(n-1)] */

```

Figure 13. Partitioning algorithm (extended collision method – three-way)

	Sweep simple		Classic collision		Collision new	
n	C_n^{avg}	M_n^{avg}	C_n^{avg}	M_n^{avg}	C_n^{avg}	M_n^{avg}
2	1.0	5.5	2.500	4.000	1.5	5.0
3	2.0	7.0	3.333	4.500	2.5	5.5
4	3.0	8.5	4.917	4.750	3.5	6.0
5	4.0	10.0	6.200	5.200	4.5	6.5
6	5.0	11.5	7.300	5.600	5.5	7.0
7	6.0	13.0	8.381	6.071	6.5	7.5
8	7.0	14.5	9.423	6.518	7.5	8.0
9	8.0	16.0	10.460	7.000	8.5	8.5
10	9.0	17.5	11.483	7.467	9.5	9.0
11	10.0	19.0	12.505	7.955	10.5	9.5
12	11.0	20.5	13.520	8.432	11.5	10.0
13	12.0	22.0	14.534	8.923	12.5	10.5
14	13.0	23.5	15.544	9.407	13.5	11.0
15	14.0	25.0	16.554	9.900	14.5	11.5

Table 1. Average numbers of comparisons C_n^{avg} and data movements M_n^{avg} , needed by three different partitioning algorithms. Values have been obtained by counting over all permutations of n pairwise different elements. Partitioning elements are always taken from index $\lfloor n/2 \rfloor$. The algorithms are listed in Figures 10, 12, and 1, respectively.

	Sweep extended		Classic collision ext.		Bentley & McIlroy	
n	C_n^{avg}	M_n^{avg}	C_n^{avg}	M_n^{avg}	C_n^{avg}	M_n^{avg}
2	3.5	5.5	3.500	2.500	4.500	7.0
3	5.0	7.0	5.667	3.500	6.167	7.5
4	6.5	8.5	7.333	4.250	7.833	8.0
5	8.0	10.0	8.950	4.900	9.500	8.5
6	9.5	11.5	10.533	5.500	11.167	9.0
7	11.0	13.0	12.057	6.143	12.833	9.5
8	12.5	14.5	13.613	6.679	14.500	10.0
9	14.0	16.0	15.111	7.321	16.167	10.5
10	15.5	17.5	16.653	7.825	17.833	11.0
11	17.0	19.0	18.144	8.464	19.500	11.5
12	18.5	20.5	19.676	8.950	21.167	12.0
13	20.0	22.0	21.167	9.581	22.833	12.5
14	21.5	23.5	22.693	10.058	24.500	13.0
15	23.0	25.0	24.186	10.681	26.167	13.5

Table 2. Average numbers of comparisons C_n^{avg} and data movements M_n^{avg} , needed by three different three-way partitioning algorithms. Values have been obtained by counting over all permutations of n pairwise different elements. Partitioning elements are always taken from index $\lfloor n/2 \rfloor$. Algorithms “sweep extended” and “classic collision extended” are listed in Figures 11 and 13, respectively.

The following table shows the frequencies of comparisons obtained by counting over all permutations of $n = 10$ pairwise different elements for two three-way algorithms (partitioning elements taken from index $\lfloor n/2 \rfloor$). In both cases the distribution appears irregular:

Comparisons	Classic collision ext.	Bentley & McIlroy
12	403,200	25,920
13	141,120	2,880
14	413,280	285,120
15	252,000	118,080
16	806,400	622,080
17	161,280	535,680
18	241,920	518,400
19	403,200	679,680
20	403,200	362,880
21	403,200	478,080

The first decision when choosing an algorithm should be that between two-way and three-way partitioning. A three-way version will need, on average, about 50% more comparisons than a two-way method. In a Quicksort implementation, this only pays off if the number of distinct values of elements is small compared to n . For our numerical analysis we therefore decided for a two-way version.

Second, sweep methods should be dropped for practical use because of their poor performance. From the remaining two-way algorithms that we discussed, only the “new collision” method has properties (equations (3)) to make it suitable for our analysis. Because of its optimal partitioning—always arrangement (1) shown in Section 2.1—it should also be favored in practice.

If the decision is for a three-way version and if comparisons of elements by operators are done directly within the sorting program, we suggest to use the “extended classic collision” method for efficiency reasons. The advantage of Bentley and McIlroy’s version for polymorphic programs like `qsort` has already been pointed out above.

8. Our Quicksort implementation

In the appendix, we present a Quicksort program conforming to the C99 standard [27] and compatible also with C++. It incorporates an implementation of the recursive median of three medians method and is meant to reduce the probability of bad cases to very small values which can be regarded as virtually zero. In contrast to the C library function `qsort`, this version is not polymorphic, so the type of array elements has to be fixed before compilation.

Parameterizations are used to support easy adaptation to the needs of application programs. The type of array elements is declared in a single `typedef` statement. Comparison operators for the element type are defined as preprocessor macros.

The values of parameters q_{\min} (equation (33)) and n_{\max}^b (Section 4.5) are set in `enum` statements for identifiers `QMIN` and `NBASISMAX`, respectively. Decreasing the value for `QMIN` will decrease the probability for bad cases, but will increase the average sorting time. We recommend a value within the range 5 to 10. Ideally, the optimum value for `NBASISMAX` would then be determined by time measurements. In most cases, a value within the range 10 to 20 turns out to be reasonable.

The switch `THREEWAY` allows to choose between two-way and three-way partitioning. The respective algorithms are those shown in Figures 1 and 13. The three-way version will, on average, need about 50% more comparisons for partitioning than its two-way counterpart. This is a waste of resources if the ratio n_d/n , where n_d is the number of distinct values of elements, is near to one. So if this characteristic of data is known for an application, the two-way version should be preferred, whereas for a general purpose sorting function the three-way variant is the method of choice.

For the median of three selection we used, apart from the element in the middle of the array, those at index positions $\lfloor n/4 \rfloor$ and $\lfloor n/2 \rfloor + \lfloor n/4 \rfloor$ instead of the first and last ones. It turned out that due to an

interference with our partitioning method, the usual version leads to a (nearly) worst case behavior for decreasing element values, which is unacceptable for our program.

Recursive calls of the sort function have been replaced by loops and use of an explicit stack. This well-known technique was first published by Hibbard [13]. However, determination of the recursive median of medians is done by a recursive function, which uses sample size $m = 9$ as the only basis case. The recursion depth for such a determination is $\lfloor \log_3 (n/q_{\min}) \rfloor - 2$.

Time measurements

Time measurements for our implementation were carried out on a desktop PC, installed in 2014 and running under Windows 7. We used the C compiler from Microsoft Visual Studio 2012[†], always with optimization switch /O2. Installation parameters for Quicksort were set—where applicable—as QMIN=5 and NBASISMAX=15.

Measurements were done for data generated as uniformly distributed (pseudo) random, strictly increasing, strictly decreasing, all equal, having organ pipe characteristic, and uniformly distributed (pseudo) random sequences of only two values. Organ pipe characteristic means that values are strictly increasing up to the middle of the array and then, as mirror image, strictly decreasing up to the end.

Results are shown in Table 3. Additionally to our favored version that implements model 5, we present results for models 1, 2, and 3, all of them with two-way and three-way partitioning, and for two variants of Heapsort.

The table shows results for integer data (`int`, 4 bytes), supplemented by those for random sequences of double precision floating point (`double`, 8 bytes) and record data (32 bytes), the latter of a type declared as

```
typedef struct {
    int key;
    int data[7];
} Record;
```

Already at first glance, one can see that for `int`-arrays, Quicksort needed sorting times below 0.1 seconds for up to one million elements, and below 0.5 seconds for five million elements, with the only exception of the pathological case organ pipe with model 1. A similar anomaly would have shown up with model 2 (median of three) for both decreasing values and organ pipe, had we not made the modification mentioned above.

A further look confirms that three-way partitioning has its cost compared to the two-way version except for cases where the number of distinct values of elements is small compared to n . In our tests these are the cases of all equal values and of random data of only two different ones, where three-way partitioning exhibits its power. Second, a greater effort to select the partitioning element obviously leads to costs higher than could be expected from our results for numbers of comparisons, which were also confirmed by simulations. Several reasons may be responsible for this, such as the overhead of organizing the selection of the partitioning element, and the non-locality of memory accesses in this connection. It should be kept in mind, however, that our main objective is to reduce the probability of bad cases, not to reduce the average sorting time.

We also made time measurements with two Heapsort variants: the original version (Williams [28]), labelled “classic” in Table 3, and Bottom-Up Heapsort (Wegener [29]), labelled “BU”. These methods are known to have worst case time complexity $O(n \log n)$ and, as does Quicksort, they sort in place, but in contrast to Quicksort do not need additional stack space.

Comparing the sorting times of these two versions, one can see that the bottom-up variant needs longer for small values of n , but—at least for random and sorted data of distinct values—the time is increasing more slowly with n , so that this version eventually overtakes the classic variant in speed. In the test cases where the number of distinct values of elements is small compared to n , however, the classic version turns out to be the faster one.

[†]Surprisingly, the compiler does not support use of the x86 extended precision data type that we need for our numerical computations.

n	Model 1		Model 2		Model 3		Model 5		Heapsort	
	2-way	3-way	2-way	3-way	2-way	3-way	2-way	3-way	classic	BU
int random										
100,000	5.6	6.7	5.9	6.8	6.0	7.1	6.2	6.9	5.5	7.3
500,000	31.5	37.3	32.8	38.2	33.8	39.7	35.4	39.2	41.8	44.7
1,000,000	66.0	77.9	69.0	79.6	70.7	82.8	75.1	83.0	101.1	97.7
5,000,000	368.9	431.7	378.6	439.3	391.6	453.8	425.3	468.6	1121.8	802.6
int increasing										
100,000	0.9	1.4	1.0	1.5	1.0	1.5	0.9	1.4	4.2	4.4
500,000	4.8	7.6	5.1	7.9	5.2	8.1	5.2	7.8	24.3	24.0
1,000,000	10.1	15.9	10.6	16.4	10.9	16.9	11.1	16.6	51.0	49.4
5,000,000	59.5	91.2	64.4	93.6	64.4	95.1	68.6	98.1	312.9	282.2
int decreasing										
100,000	1.0	1.5	1.1	1.5	1.1	1.5	1.0	1.4	4.0	4.8
500,000	5.6	7.7	5.8	8.0	5.7	8.2	5.6	7.9	23.4	25.8
1,000,000	11.6	16.1	12.1	16.7	11.9	17.2	11.8	16.8	51.9	53.9
5,000,000	70.5	93.4	68.4	95.4	67.7	96.7	73.1	99.6	302.2	301.8
int equal										
100,000	1.5	0.1	1.5	0.1	1.5	0.1	1.3	0.1	0.3	2.4
500,000	7.9	0.6	8.1	0.6	8.0	0.6	7.2	0.6	1.6	15.1
1,000,000	16.6	1.1	17.0	1.1	16.7	1.1	15.2	1.3	3.3	38.0
5,000,000	92.7	5.7	94.8	5.9	94.5	5.9	98.0	8.1	16.9	255.3
int organ pipe										
100,000	20.4	44.6	2.1	2.3	2.6	2.9	2.2	2.6	4.2	5.2
500,000	803	3381	11.3	12.8	13.6	15.4	11.6	14.0	24.8	29.4
1,000,000	3172	3560	23.7	26.3	27.9	31.8	23.9	29.0	53.3	61.4
5,000,000	96539	69761	127.2	143.4	149.8	172.2	134.5	163.1	334.9	361.0
int {0, 1} random										
100,000	1.8	0.4	1.9	0.5	1.9	0.4	1.8	0.5	2.5	3.5
500,000	9.9	2.2	10.1	2.3	10.2	2.2	10.1	2.5	14.1	19.3
1,000,000	20.5	4.5	21.0	4.5	21.0	4.5	21.2	5.2	29.3	41.6
5,000,000	112.9	22.9	114.6	23.4	114.9	23.2	118.1	27.3	170.2	254.6
double random										
100,000	6.2	7.4	6.4	7.5	6.7	7.7	6.9	7.6	7.2	8.1
500,000	34.8	41.2	36.2	42.1	37.9	43.2	39.4	43.4	54.5	49.9
1,000,000	73.2	86.8	76.0	88.4	79.4	90.7	84.3	92.8	144.2	119.9
5,000,000	408.0	487.9	412.6	486.9	441.1	505.8	487.9	525.7	1661.1	1056.1
Record (32 bytes) random										
100,000	6.7	7.8	7.0	8.0	7.2	8.1	7.4	8.0	10.7	10.8
500,000	42.2	49.5	42.9	49.4	43.5	50.4	46.5	50.9	118.9	97.4
1,000,000	91.9	107.7	92.9	107.3	93.8	109.5	102.0	113.3	324.1	247.3
5,000,000	545.1	662.7	545.6	646.1	545.7	642.5	612.0	678.8	2934.0	1983.2

Table 3. Results of time measurements on a PC. Sorting times in milliseconds. For details see text page 22.

In all our test cases, the Quicksort version corresponding to model 5 proved to be faster than Bottom-Up Heapsort, though the difference might not always appear really impressive. For random data, the lead of Quicksort is increasing with increasing n . This effect even gets larger for data elements of greater size. Further, the ability of Quicksort, notably the three-way version, to adapt itself to characteristics of data is much more distinct than that of Heapsort.

If, as usual in theoretical work (and as was our procedure too), the number of comparisons is taken as measure for the cost of a sorting algorithm, Wegener's claim that he made in his article [29] (see its title) is correct. As we could see from results of simulation programs, the numbers of comparisons of elements for random data in the range $n = 100,000$ to five millions are actually smaller for Bottom-Up Heapsort than for all our versions of Quicksort. However, on real-world computers—at least those that are in widespread use today—the locality of memory access or its absence obviously influences the performance of programs to an unexpected extent.

9. Conclusion

A few years after the fiftieth anniversary of Quicksort, we are able to present numerical results for the probability of bad cases for the algorithm. By means of simulations, such probabilities can be determined only for small numbers of elements. So we computed the frequency distributions of comparisons by solving the respective recurrence equations numerically, applying well-known techniques. From these distributions, probability values for bad cases could be extracted.

For technical reasons, we had to limit our computations to numbers of elements not greater than $n = 500$. We believe, however, that even results in this limited range allow some recognition of the probabilities we are interested in.

So, how many cases are bad cases? That depends, of course, on how bad cases are exactly defined, on the number of elements to be sorted, and on the Quicksort version used. Probability values and their dependencies are presented in Section 6.4 by means of figures and tables.

The large influence of the selection method on the probability of bad cases encouraged us to investigate an extension of the median of three medians selection, which we call “recursive median of three medians.” In order to prove that it is applicable in practice, we developed the implementation that is listed in the appendix. This version not only reduces probabilities of bad cases to extremely small values and leads to a worst case complexity below $O(n^2)$, but also helps to prevent that “non-random” data, i. e. sequences already possessing some order, are preferred candidates for bad cases. Time measurements showed that this version of Quicksort is faster than Heapsort and Bottom-Up Heapsort for large numbers of elements. For Bottom-Up Heapsort and random data, this is in contrast to average numbers of comparisons which we determined by simulations. Theory is important, but is always an abstraction from reality, so tests are necessary to confirm or disprove the applicability in practice.

Source files containing the Quicksort version listed in the appendix and our version of `qsort` are submitted to the arXiv together with this article.

ACKNOWLEDGEMENTS

This work has been done as a private research project of the author after retirement from teaching. I am indebted to the Government of the Land Schleswig-Holstein, Germany, and to our university for providing me with the necessary computing resources. Finally, I would like to thank our laboratory engineers Rolf Himmighoffen and Maike Sieloff for their technical assistance.

References

- [1] Hoare, C. A. R. *Algorithm 63: Partition. Algorithm 64: Quicksort.* Communications of the ACM 4 (1961), 321.
- [2] Hoare, C. A. R. *Quicksort.* The Computer Journal 5 (1962), 10–15.

- [3] McDiarmid, C. J. H., Hayward, R. B. *Strong concentration for Quicksort*. SODA '92 Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms, Orlando, Florida (1992), 414–421.
- [4] McDiarmid, C. J. H., Hayward, R. B. *Large Deviations for Quicksort*. Journal of Algorithms 21 (1996), 476–507.
- [5] Fill, J. A., Janson, S. *Quicksort asymptotics*. Journal of Algorithms 44 (2002), 4–28.
- [6] Eddy, W. F., Schervish, M. J. *How Many Comparisons Does Quicksort Use?* Journal of Algorithms 19 (1995), 402–431.
- [7] Sedgewick, R. *Algorithms in C++. Parts 1–4*. Third Edition. Addison Wesley, Reading, Massachusetts, 1998.
- [8] Singleton, R. C. *An Efficient Algorithm for Sorting with Minimal Storage*. Communications of the ACM 12 (1969), 185–187.
- [9] Bentley, J. L., McIlroy, M. D. *Engineering a Sort Function*. Software—Practice and Experience 23 (1993), 1249–1265.
- [10] Durand, M. *Asymptotic analysis of an optimized quicksort algorithm*. Information Processing Letters 85 (2003) 73–77.
- [11] Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., Tarjan, R. E. *Time Bounds for Selection*. Journal of Computer and System Sciences 7 (1973), 448–461.
- [12] Hillmore, J. S. *Certification of Algorithms 63, 64, 65 Partition Quicksort Find*. Communications of the ACM 5 (1962), 439.
- [13] Hibbard, T. N. *An Empirical Study of Minimal Storage Sorting*. Communications of the ACM 6 (1963), 206–213.
- [14] McIlroy, M. D. *A Killer Adversary for Quicksort*. Software—Practice and Experience 29 (1999), 1–4.
- [15] Knuth, D. E. *The Art of Computer Programming. Vol. 3: Sorting and Searching*. Second Edition. Addison Wesley, Reading, Massachusetts, 1998.
- [16] Sedgewick, R., Flajolet, P. *An Introduction to the Analysis of Algorithms*. Addison Wesley, Boston, 1996.
- [17] Iliopoulos, V., Penman, D. *Variance of the number of Comparisons of Randomised Quicksort*. arXiv:1006.4063 [math.PR] 2010. (<http://arxiv.org>)
- [18] Kernighan, B. W., Ritchie, D. M. *The C Programming Language*. Second Edition. Prentice Hall, Englewood Cliffs, 1988.
- [19] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms*. Third Edition. MIT Press, Cambridge, Massachusetts, 2009.
- [20] Bentley, J. *Programming Pearls*. Second Edition. Addison Wesley, Boston, 2000.
- [21] Scowen, R. S. *Quickersort*. Communications of the ACM 8 (1965), 669–670.
- [22] van Emden, M. H. *Increasing the Efficiency of Quicksort*. Communications of the ACM 13 (1970), 563–567.
- [23] Wirth, N. *Algorithmen und Datenstrukturen. Pascal-Version*. Teubner, Stuttgart, 1975 (Fifth Edition 2000).
- [24] Wirth, N. *Algorithms & Data Structures*. Prentice Hall, Upper Saddle River, NJ, 1986.
- [25] Sedgewick, R. *Algorithms in C*. Addison Wesley, Reading, Massachusetts, 1990.
- [26] Sedgewick, R. *Implementing Quicksort Programs*. Communications of the ACM 21 (1978), 847–857.
- [27] International Standard ISO/IEC 9899. *Programming languages – C*. Second edition. 1999.
- [28] Williams, J. W. J. *Algorithm 232: Heapsort*. Communications of the ACM 7 (1964), 347–348.
- [29] Wegener, I. *BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small)*. Theoretical Computer Science 118 (1993), 81–98.

Appendix: Listing of a Quicksort implementation in C

```

/*****
*
* Name:          quicksort
*
* Purpose:       Sorting of an array of element type int into
*               nondescending order.
*
* Prototype:     void quicksort(size_t n, int a[]);
*
* Parameters:
*
* n:             (in) Number of array elements to be sorted.
*
* a:             (in/out) On input, the array to be sorted.
*               On output, the sorted array.
*
* Method:        Sorting is done by nonrecursive Quicksort, using a
*               stack to store (sub)problems yet to be solved.
*               For the selection of partitioning elements, a recursive
*               median of three medians approach is used. Sorting of
*               small (sub)arrays is done by insertion sort.
*****/
#include <stddef.h>
#include <limits.h>

#define THREWAY 1 /* != 0: Use three-way partitioning */

/*-----*/
typedef int Elemtyp;

/* Comparison operators for Elemtyp: */
#define LT(v1, v2) ((v1) < (v2)) /* less than */
#define LE(v1, v2) ((v1) <= (v2)) /* less than or equal */
#define EQ(v1, v2) ((v1) == (v2)) /* equal */
/*-----*/

static size_t medofmed(size_t m, size_t inc, const Elemtyp a[]);

void quicksort(size_t n, Elemtyp a[])
{
/*-----*/
    enum {QMIN=5}; /* For recursive median of three medians:
                   minimum value for (n / sample size)
                   (must be >= 1) */
    enum {NBASISMAX=15}; /* Maximum (sub)array size for insertion sort
                        (must be >= 3) */
/*-----*/
    enum {FALSE, TRUE};
    enum {STACKSIZE=sizeof(size_t)*CHAR_BIT*2};
    int top, done;
    size_t left, right, nt, i, j, k, ipartel, nc, m, mt;
    Elemtyp partval, temp;
    size_t stack[STACKSIZE];

```

```

if (n <= 1)
    top = 0;
else {
    stack[0] = 0;
    stack[1] = n-1;
    top = 2;
}

/* loop to deal with stacked intervals
yet to be sorted */
while (top > 0) {
    right = stack[--top];
    left = stack[--top];

    /* loop to replace tail recursion */
    while (left < right) {
        nt = right-left+1;
        if (nt <= NBASISMAX) { /* use insertion sort */

            for (i = left+1; i <= right; i++) {
                if (LT(a[i], a[j=i-1])) {
                    temp = a[i]; a[i] = a[j];
                    while (j > left && LT(temp, a[k=j-1])) {
                        a[j] = a[k]; j = k;
                    }
                    a[j] = temp;
                }
            }
            left = right; /* to terminate loop */
        }
    } else {

        if (nt < (QMIN*9)) { /* median of three */

            i = left+nt/4; k = left+nt/2; j = k+nt/4;
            ipartel = (LE(a[i], a[k]) ?
                (LE(a[k], a[j]) ? k :
                    LT(a[i], a[j]) ? j : i) :
                (LE(a[j], a[k]) ? k :
                    LT(a[j], a[i]) ? j : i));
        } else {
            /* recursive median of medians */
            nc = nt/(QMIN*9);
            m = 1;
            while ((mt = m*3) <= nc)
                m = mt;
            m *= 9; /* m is sample size */

            ipartel = medofmed(m, (nt-1)/(m-1), a+left) + left;
        }

        partval = a[ipartel];
#ifdef !THREEWAY /* two-way partitioning */
        a[ipartel] = a[right];
        a[right] = partval;
        i = left;
        j = right-1;
        done = FALSE;
        do {
            while (LT(a[i], partval))
                i++;

```

```

        while (i < j && LT(partval, a[j]))
            j--;
        if (j <= i)
            done = TRUE;
        else {
            temp = a[i]; a[i++] = a[j]; a[j--] = temp;
        }
    } while (!done);
    a[right] = a[i];
    a[i] = partval;
    j = i;

#else
                                /* three-way partitioning */
    j = left;
    k = right;
    done = FALSE;
    do {
        while (LT(a[j], partval))
            j++;
        while (j < k && LE(partval, a[k]))
            k--;
        if (k <= j)
            done = TRUE;
        else {
            temp = a[j]; a[j++] = a[k]; a[k--] = temp;
        }
    } while (!done);

    k = j;
    i = right;
    done = FALSE;
    do {
        while (k <= i && EQ(partval, a[k]))
            k++;
        while (k <= i && LT(partval, a[i]))
            i--;
        if (i <= k)
            done = TRUE;
        else {
            temp = a[k]; a[k++] = a[i]; a[i--] = temp;
        }
    } while (!done);

#endif
                                /* push boundaries of greater
                                subarray onto stack */
    if (j-left <= right-i) {
        stack[top++] = i+1;
        stack[top++] = right;
        right = (j == 0) ? j : j-1;
    } else {
        stack[top++] = left;
        stack[top++] = (j == 0) ? j : j-1;
        left = i+1;
    }
}
}
}
}

```

```

/*****
*
* Name:          medofmed
*
* Purpose:       Determination of the recursive median of three medians
*                from a sample of elements of type int.
*
* Prototype:     size_t medofmed(size_t m, size_t inc, const int a[]);
*
* Parameters:
*
* m:             (in) Number of array elements to be used as sample.
*                The value must be a power of 3 and >= 9.
*
* inc:           (in) Increment of array elements to be used as sample.
*
* a:             (in) Array containing elements to be used as sample.
*
* Return value:  Index of element containing the recursive median of
*                three medians.
*
*****/
static size_t medofmed(size_t m, size_t inc, const Elemtyp a[])
{
#define MEDOF3(i, j, k) \
    (LE(a[i], a[j]) ? \
     (LE(a[j], a[k]) ? j : LT(a[i], a[k]) ? k : i) : \
     (LE(a[k], a[j]) ? j : LT(a[k], a[i]) ? k : i))

    size_t i0, i1, i2;

    if (m == 9) {
        /* recursion basis */
        size_t inc3 = inc*3;
        size_t i = 0, j = inc, k = j+inc;
        i0 = MEDOF3(i, j, k);
        i += inc3; j += inc3; k += inc3;
        i1 = MEDOF3(i, j, k);
        i += inc3; j += inc3; k += inc3;
        i2 = MEDOF3(i, j, k);
    } else {
        /* recursive part */
        m /= 3;
        i0 = medofmed(m, inc, a);
        i1 = medofmed(m, inc, a+m*inc) + m*inc;
        i2 = medofmed(m, inc, a+m*inc*2) + m*inc*2;
    }

    return MEDOF3(i0, i1, i2);
}

```